J. Venkat Vardhan
200968158
Roll No: 41

Problem Statement:

Frequent, and increasingly severe, natural disasters threaten human health, infrastructure, and natural systems. The provision of accurate, timely, and understandable information has the potential to revolutionize disaster management. For quick response and recovery on a large scale, after a natural disaster such as a hurricane, access to aerial images is critically important for the response team. The emergence of small unmanned aerial systems (UAS) along with inexpensive sensors presents the opportunity to collect thousands of images after each natural disaster with high flexibility and easy manoeuvrability for rapid response and recovery. Moreover, UAS can access hard-to-reach areas and perform data collection tasks that can be unsafe for humans if not impossible. Despite all these advancements and efforts to collect such large datasets, analysing them and extracting meaningful information remains a significant challenge in scientific communities.

Approach:

FloodNet provides high-resolution aerial imageries with detailed semantic annotation regarding the damages caused by these natural disasters. We propose a solution to address the image classification and semantic segmentation challenge. We approach this problem by generating pseudo labels for both classification and segmentation during training and slowly incrementing the amount by which the pseudo label loss affects the final loss. Using this semi-supervised method of training helped us improve our baseline supervised loss by a huge margin for classification, allowing the model to generalize and perform better on the validation and test splits of the dataset. In this paper, we compare and contrast the various methods and models for image classification and semantic segmentation on the FloodNet dataset.

Meta-Data:

There are 3 separate datasets, Train, Test and Validation. The dataset is classifying into ‘Flooded’ and ‘Non-Flooded’ classes. Only a few of the training images have their labels available, while most of the training images are unlabelled. The labelling part will be taken care by the Semi-Supervised Classification.

The second part which involves the Semi-Supervised Semantic Segmentation assign labels like 1) Background, 2) Building Flooded, 3) Building Non-Flooded, 4) Road Flooded, 5) Road Non-Flooded, 6) Water, 7)Tree, 8) Vehicle, 9) Pool, 10) Grass.

Objectives:

1. To get a detailed post-disaster damage assessment
2. To spot disaster-prone areas
3. To make faster semantic analysis of images is crucial for rational rescue plans and resource allocation
4. Identification of flood-prone area to avoid constriction of infrastructures or houses

which exceeds the floor limit.

Models:

1. Convolutional Neural Networks (CNNs)
2. Long Short Term Memory Networks (LSTMs)
3. Recurrent Neural Networks (RNNs)
4. Generative Adversarial Networks (GANs)
5. Radial Basis Function Networks (RBFNs)
6. Multilayer Perceptrons (MLPs)
7. Deep Belief Networks (DBNs)
8. Autoencoders

CNNs are used an input dataset is if spatial structure like an image etc.
RNNS, LSTM, GAN, RBFN are used when it's an sequential structure.
MLP or fully connected NN are used when there is no structure.

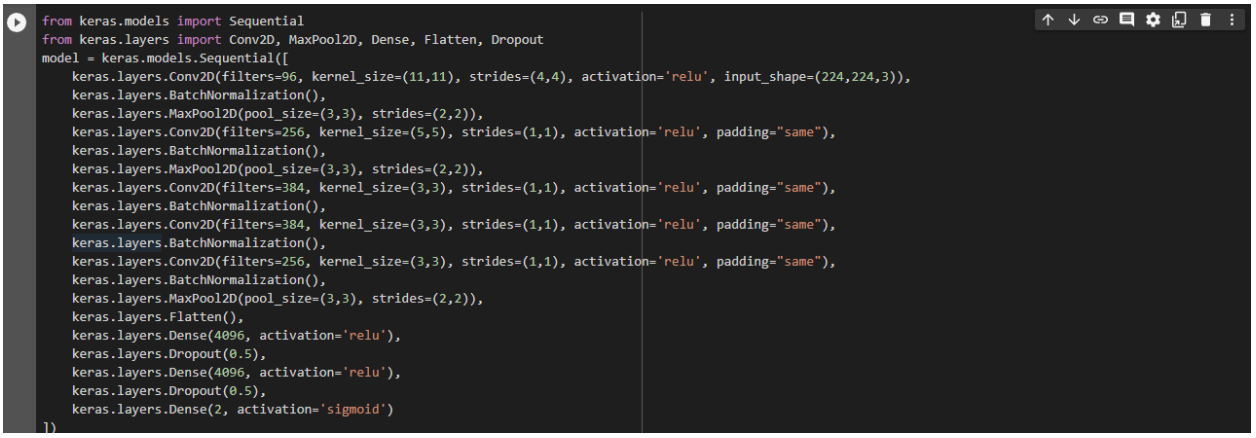
So, as our FloodNet dataset is of satellite images, we will be adopting CNNs as our model and under CNN, we can use AlexNet, VGGNet, UNet, PSPNet, ParseNet, DeepLabV etc.

Baseline Model:

Our baseline models are Convolutional Neural Networks (CNNs)

The models used here are AlexNet and VGGNet.

Architecture:



```
from keras.models import Sequential
from keras.layers import Conv2D, MaxPool2D, Dense, Flatten, Dropout
model = keras.models.Sequential([
    keras.layers.Conv2D(filters=96, kernel_size=(11,11), strides=(4,4), activation='relu', input_shape=(224,224,3)),
    keras.layers.BatchNormalization(),
    keras.layers.MaxPool2D(pool_size=(3,3), strides=(2,2)),
    keras.layers.Conv2D(filters=256, kernel_size=(5,5), strides=(1,1), activation='relu', padding="same"),
    keras.layers.BatchNormalization(),
    keras.layers.MaxPool2D(pool_size=(3,3), strides=(2,2)),
    keras.layers.Conv2D(filters=384, kernel_size=(3,3), strides=(1,1), activation='relu', padding="same"),
    keras.layers.BatchNormalization(),
    keras.layers.Conv2D(filters=384, kernel_size=(3,3), strides=(1,1), activation='relu', padding="same"),
    keras.layers.BatchNormalization(),
    keras.layers.Conv2D(filters=256, kernel_size=(3,3), strides=(1,1), activation='relu', padding="same"),
    keras.layers.BatchNormalization(),
    keras.layers.MaxPool2D(pool_size=(3,3), strides=(2,2)),
    keras.layers.Flatten(),
    keras.layers.Dense(4096, activation='relu'),
    keras.layers.Dropout(0.5),
    keras.layers.Dense(4096, activation='relu'),
    keras.layers.Dropout(0.5),
    keras.layers.Dense(2, activation='sigmoid')
])
```

The optimizer used here is “Adam”, loss function is categorical cross entropy and the metric used here is accuracy.

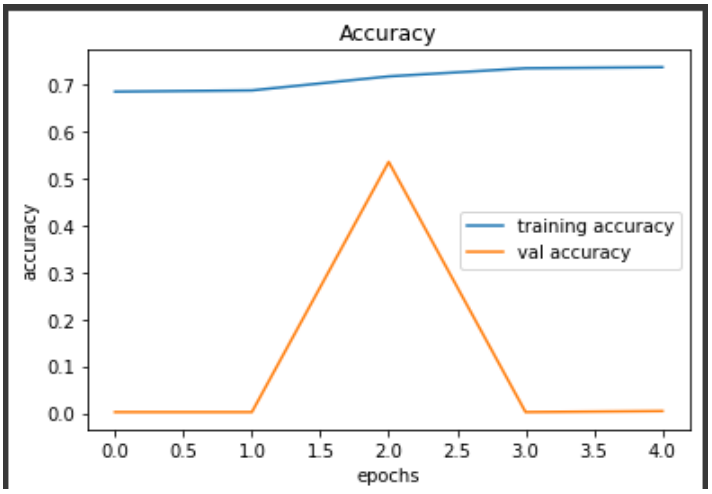
Training:

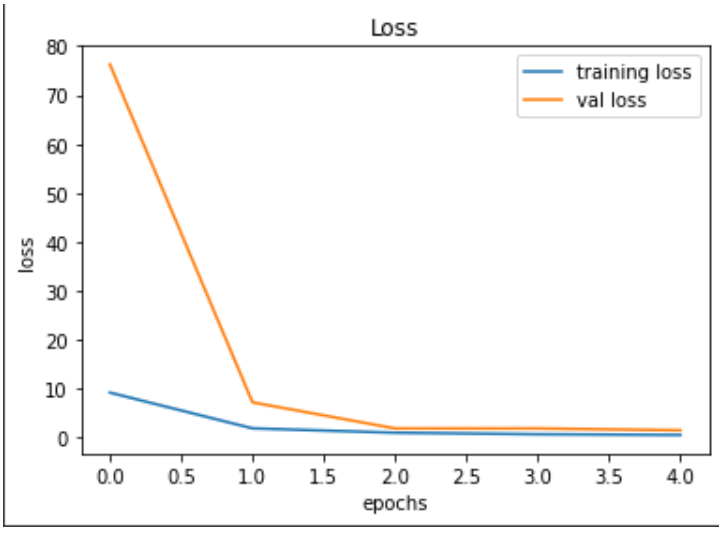
```
[17] model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])

[18] history = model.fit(train_generator, batch_size=32, epochs=5, validation_data=validation_generator)

     Epoch 1/5
     93/93 [==============================] - 1567s 17s/step - loss: 9.2230 - accuracy: 0.6853 - val_loss: 76.2595 - val_accuracy: 0.0022
     Epoch 2/5
     93/93 [==============================] - 795s 9s/step - loss: 1.9136 - accuracy: 0.6880 - val_loss: 7.2272 - val_accuracy: 0.0022
     Epoch 3/5
     93/93 [==============================] - 803s 9s/step - loss: 1.0248 - accuracy: 0.7179 - val_loss: 1.8991 - val_accuracy: 0.5356
     Epoch 4/5
     93/93 [==============================] - 790s 8s/step - loss: 0.7112 - accuracy: 0.7352 - val_loss: 1.8950 - val_accuracy: 0.0022
     Epoch 5/5
     93/93 [==============================] - 784s 8s/step - loss: 0.5627 - accuracy: 0.7374 - val_loss: 1.5220 - val_accuracy: 0.0044
```

Results:

The accuracy the above model yields is approx. 0.8 .

Performance Curves:



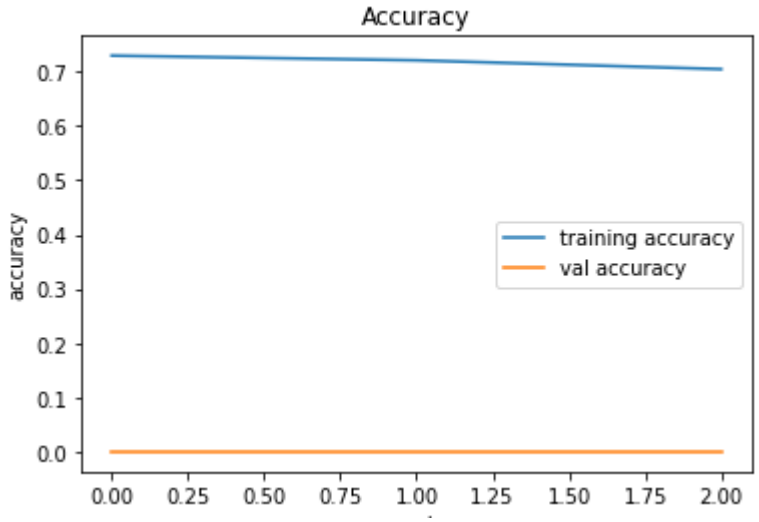


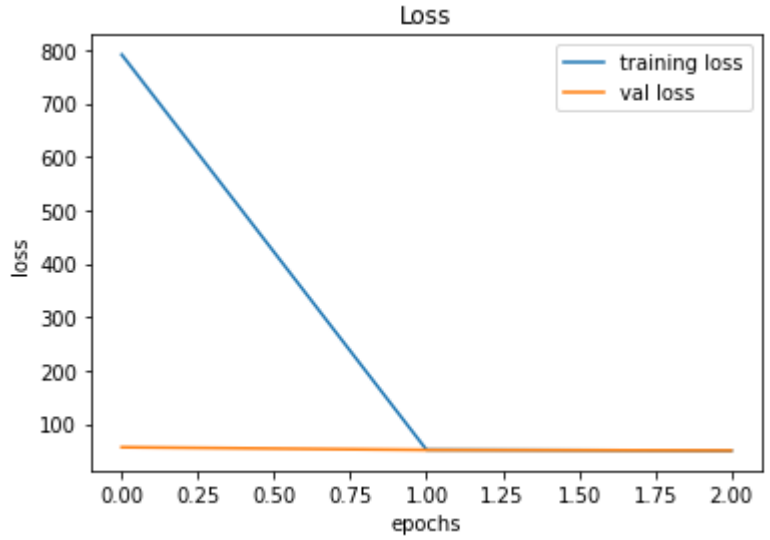
And we can make various models by adding and modifying hyperparameters like learning rate, number of hidden layers, regularization etc.

Regularization:

```
model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
history = model.fit(train_generator, batch_size=32, epochs=3, validation_data=validation_generator)
```

```
Epoch 1/3
93/93 [==============================] - 811s 9s/step - loss: 790.9061 - accuracy: 0.7298 - val_loss: 56.6204 - val_accuracy: 0.0000e+00
Epoch 2/3
93/93 [==============================] - 796s 9s/step - loss: 51.8595 - accuracy: 0.7211 - val_loss: 51.5565 - val_accuracy: 0.0000e+00
Epoch 3/3
93/93 [==============================] - 797s 9s/step - loss: 50.2094 - accuracy: 0.7048 - val_loss: 50.4734 - val_accuracy: 0.0000e+00
```

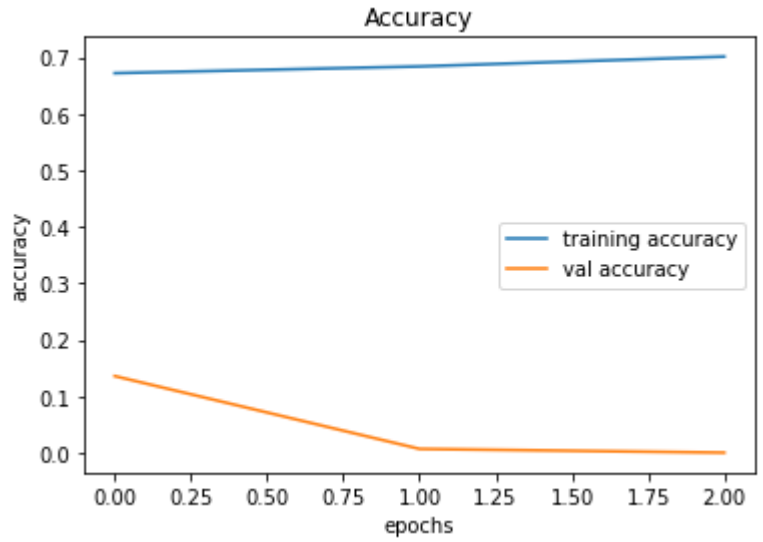


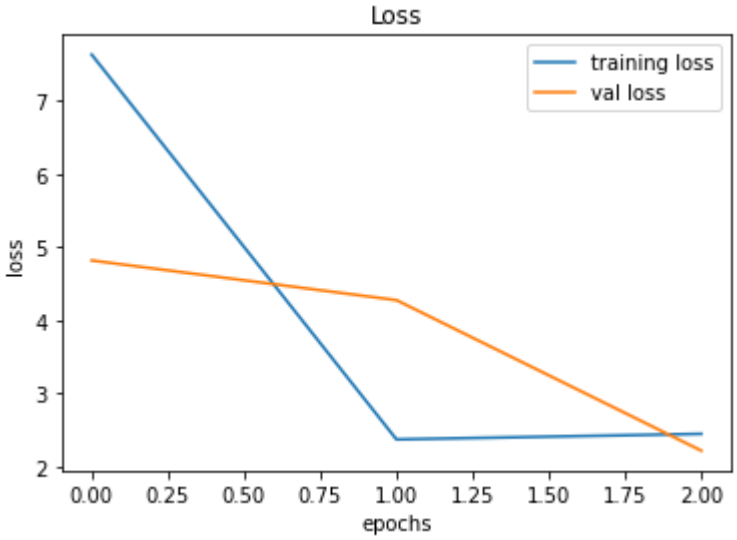


Changing learning rate:

```
model.compile(loss='categorical_crossentropy', optimizer=tf.keras.optimizers.Adam(
                    learning_rate=0.001,
                    beta_1=0.9,
                    beta_2=0.999,
                    epsilon=1e-07,
                    amsgrad=False,
                    name='Adam'
                ), metrics=['accuracy'])
history = model.fit(train_generator, batch_size=32, epochs=3, validation_data=validation_generator)
```

```
Epoch 1/3
93/93 [==============================] - 807s 9s/step - loss: 7.6306 - accuracy: 0.6717 - val_loss: 4.8154 - val_accuracy: 0.1356
Epoch 2/3
93/93 [==============================] - 780s 8s/step - loss: 2.3679 - accuracy: 0.6837 - val_loss: 4.2726 - val_accuracy: 0.0067
Epoch 3/3
93/93 [==============================] - 792s 9s/step - loss: 2.4437 - accuracy: 0.7010 - val_loss: 2.2142 - val_accuracy: 0.0000e+00
```





And in the same way, we can change the number of hidden layers in the model to detect a difference in the performance.

We can also use an all-new architecture such as VGG16 etc.

Conclusion:

The accuracy we get:

- Normally is 0.8
- Changing the regularization is 0.7048
- Changing the learning rate is 0.7010
- Changing the hidden layers is 0.702

So, we can conclude that by using the AlexNet architecture, even by updating the hyperparameters, the accuracy in each case model is more or less the same, the difference is in points. So, the architecture used above yields good results and is very efficient.